université de BORDEAUX

Formation CNRS
18 Novembre 2016
Python pour la biologie

Biopython

cgfb
BIOINFORMATIQUE

université de BORDEAUX

Qu’est ce que Biopython ?

- The Biopython Project is an international association of developers of freely available Python tools for computational molecular biology

- The Biopython web site (http://www.biopython.org) provides an online resource for modules, scripts, and web links for developers of Python-based software for bioinformatics use and research.

- Basically, the goal of Biopython is to make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and classe

Les fonctionnalités Biopython (1)

- The ability to parse bioinformatics files into Python utilizable data structures, including support for the following formats:
  - Blast output - both from standalone and WWW Blast
  - Clustalw
  - FASTA
  - GenBank
  - PubMed and Medline
  - ExPASy files, like Enzyme and Prosite
  - SCOP, including “dom” and “lin” files
  - UniGene
  - SwissProt

Les fonctionnalités Biopython (2)

- Files in the supported formats can be iterated over record by record or indexed and accessed via a Dictionary interface.

- Code to deal with popular on-line bioinformatics destinations such as:
  - NCBI { Blast, Entrez and PubMed services
  - ExPASy { Swiss-Prot and Prosite entries, as well as Prosite searches

- Interfaces to common bioinformatics programs such as:
  - Standalone Blast from NCBI
  - Clustalw alignment program
  - EMBOSS command line tools

Les fonctionnalités Biopython (3)

- Standard sequence class that deals with sequences, ids on sequences, and sequence features.
- Performing common operations on sequences, such as translation, transcription and weight calculations.
- Perform classication of data using k-Nearest Neighbors, Naive Bayes or Support VectorMachines.
- Dealing with alignments, including a standard way to create and deal with substitution matrices.
- Making it easy to split up parallelizable tasks into separate processes.
- GUI-based programs to do basic sequence manipulations, translations, BLASTing, etc.
- Extensive documentation and help with using the modules, including this le, on-line wiki documentation, the web site, and the mailing list.

The Lady Slipper Orchids case

- The Lady Slipper Orchids are in the Orchidaceae family and the Cypripedioideae sub-family and are made up of 5 genera:
- Cypripedium
- Paphiopedilum
- Phragmipedium
- Selenipedium
- Mexipedium

Working with sequence: The Seq Object

- Most of the time when we think about sequences we have in my mind a string of letters like `AGTACACTGGT'.
- You can create such Seq object with this sequence as follows the ">>>" represents the Python prompt followed by what you would type in
- DON'T FORGET TO USE => from Bio.Seq import Seq in your script

```
>>>from Bio.Seq import Seq
>>>my_seq = Seq("AGTACACTGGT")
>>>my_seq
Seq(AGTACACTGGT, Alphabet())
>>> print(my_seq)
AGTACACTGGT
```

```
>>>my_seq.alphabet
Alphabet()
>>>my_seq.complement()
Seq(TCATGTGACCA, Alphabet())
>>>my_seq.reverse_complement()
Seq(ACCAGTGTACT, Alphabet())
```

Sequences et Alphabet: IUPAC Alphabet for DNA, RNA and proteins

- Available alphabets for Biopython are defined in the Bio.Alphabet module.
- IUPAC (http://www.chem.qmw.ac.uk/iupac/): Bio.Alphabet.IUPAC
  - Basic IUPACProtein class
  - Additional ExtendedIUPACProtein class with Additional elements:
    - "U" (or "Sec" for selenocysteine)
    - "O" (or "Pyl" for pyrrolysine)
  - Plus the ambiguous symbols:
    - "B" (or "Asx" for asparagine or aspartic acid)
    - "Z" (or "Glx" for glutamine or glutamic acid)
    - "J" (or "Xle" for leucine isoleucine)
    - "X" (or "Xxx" for an unknown amino acid).
  - IUPACUnambiguousDNA, which provides for just the basic letters
  - IUPACAmbiguousDNA, which provides for ambiguity letters for every possible situation
  - ExtendedIUPACDNA, which allows letters for modifiedbase

Sequences et Alphabet (2)

➔ Create an ambiguous sequence with the default generic alphabet:

```
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> my_seq.alphabet
Alphabet()
```

➔ Specify the alphabet explicitly

```
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

```
>>> from Bio.Alphabet import IUPAC
>>> my_prot = Seq("AGTACACTGGT", IUPAC.protein)
>>> my_prot
Seq('AGTACACTGGT', IUPACProtein())
>>> my_prot.alphabet
IUPACProtein()
```

Sequences act like strings (1)

- Deal with Seq objects as if they were normal Python strings
- For example getting the length, or iterating over the elements:

```
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATC", IUPAC.unambiguous_dna)
>>> for index, letter in enumerate(my_seq):
... print("%i %s" % (index, letter))
0 G
1 A
2 T
3 C

>>> print(len(my_seq))
4
```

- Access elements of the sequence in the same way as for string

```
>>> print(my_seq[0]) #first letter
G
>>> print(my_seq[2]) #third letter
T
>>> print(my_seq[-1]) #last letter
G
```

Sequences act like strings (2)

- TheSeq object has a ".count()" method, just like a string. Note that this means that like a Python string, this gives a non-overlapping count:

```
>>> "AAAA".count("AA")
2
>>> Seq("AAAA").count("AA")
2
```

- For some biological uses, you may actually want an overlapping count (i.e. 3 in this trivial example). When searching for single letters, this makes no difference:

```
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPAC.unambiguous_dna)
>>> len(my_seq)
32
>>> my_seq.count("G")
9
>>> 100 * float(my_seq.count('G') + my_seq.count('C')) / len(my_seq)
46.875
```

Sequences act like strings (3)

- While you could use the above snippet of code to calculate a GC%, note that the Bio.SeqUtils module has several GC functions already built.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> from Bio.SeqUtils import GC
>>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPAC.unambiguous_dna)
>>> GC(my_seq)
46.875
```

- Note that using the Bio.SeqUtils.GC() function should automatically cope with mixed case sequences and the ambiguous nucleotide S which means G or C.

- Also note that just like a normal Python string, the Seq object is in some ways \read-only". If you need to edit your sequence, for example simulating a point mutation, look at the Section 3.12 below which talks about the MutableSeq object.

Slicing a sequence

- Let's get a slice of the sequence

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq[4:12]
Seq('GATGGGCC', IUPACUnambiguousDNA())
```

- The new object produced is another Seq object which retains the alphabet information from the original Seq object
- Get the first, second and third codons positions using “stride” (“::”) :

```
>>> my_seq[0::3]
Seq('GCTGTAGTAAG', IUPACUnambiguousDNA())
>>> my_seq[1::3]
Seq(AGGCATGCATC, IUPACUnambiguousDNA())
>>> my_seq[2::3]
Seq('TAGCTAAGAC', IUPACUnambiguousDNA())
>>> my_seq[::-1] ## Get the reverse sequence
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())
```

Turning Seq objects into strings

➔ To write to a file, or insert into a database

```
>>> str(my_seq)
'GATCGATGGGCCTATATAGGATCGAAAATCGC'
```

➔ Calling str() on a Seq object returns the full sequence as a string

➔ Python does this automatically in the print function

```
>>> print(my_seq)
GATCGATGGGCCTATATAGGATCGAAAATCGC
```

➔ Also use the Seq object directly with a %s placeholder when using the Python string formatting or interpolation operator (%)

```
>>> fasta_format_string = ">Name\n%s\n" % my_seq
>>> print(fasta_format_string)
>Name
GATCGATGGGCCTATATAGGATCGAAAATCGC
<BLANKLINE>
```

Concatenating or adding sequences

➔ Can't add sequences with incompatible alphabets, (protein and DNA)

```
>>> from Bio.Alphabet import IUPAC
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> protein_seq + dna_seq

Traceback (most recent call last):
...
TypeError: Incompatible alphabets IUPACProtein() and IUPACUnambiguousDNA()
```

➔ To do this, first give both sequences generic alphabets

```
>>> from Bio.Alphabet import generic_alphabet
>>> protein_seq.alphabet = generic_alphabet
>>> dna_seq.alphabet = generic_alphabet
>>> protein_seq + dna_seq

Seq('EVRNAKACGT', Alphabet())
```

➔ Adding a generic nucleotide seq. to an unambiguous IUPAC DNA seq.

```
>>> from Bio.Alphabet import generic_nucleotide
>>> from Bio.Alphabet import IUPAC
>>> nuc_seq = Seq("GATCGATGC", generic_nucleotide)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> nuc_seq + dna_seq

Seq(GATCGATGCACGT, NucleotideAlphabet())
```

Concatenating or adding sequences (2)

➔ Many sequences to add together:

```
>>> from Bio.Alphabet import generic_dna
>>> list_of_seqs = [Seq("ACGT", generic_dna), Seq("AACC", generic_dna), Seq("GGTT", generic_dna)]
>>> concatenated = Seq("", generic_dna)
>>> for s in list_of_seqs:
... concatenated += s
...
>>> concatenated

Seq(ACGTAACCGGTT, DNAAlphabet())
```

➔ more elegant approach using sum function with its optional start value argument

```
>>> from Bio.Alphabet import generic_dna
>>> list_of_seqs = [Seq("ACGT", generic_dna), Seq("AACC", generic_dna), Seq("GGTT", generic_dna)]
>>> sum(list_of_seqs, Seq("", generic_dna))

Seq(ACGTAACCGGTT, DNAAlphabet())
```

Changing case

➔ very useful upper and lower methods for changing the case

```
>>> from Bio.Alphabet import generic_dna
>>> dna_seq = Seq("acgtACGT", generic_dna)
>>> dna_seq
Seq('acgtACGT', DNAAlphabet())
```

```
>>> dna_seq.upper()
Seq(ACGTACGT', DNAAlphabet())
>>> dna_seq.lower()
Seq(acgtacgt, DNAAlphabet())
```

```
>>> "GTAC" in dna_seq
False
>>> "GTAC" in dna_seq.upper()
True
```

```
>>> from Bio.Alphabet import IUPAC
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> dna_seq
Seq('ACGT', IUPACUnambiguousDNA())
>>> dna_seq.lower()
Seq('acgt', DNAAlphabet())
```

Nucleotide sequences and (reverse) complements

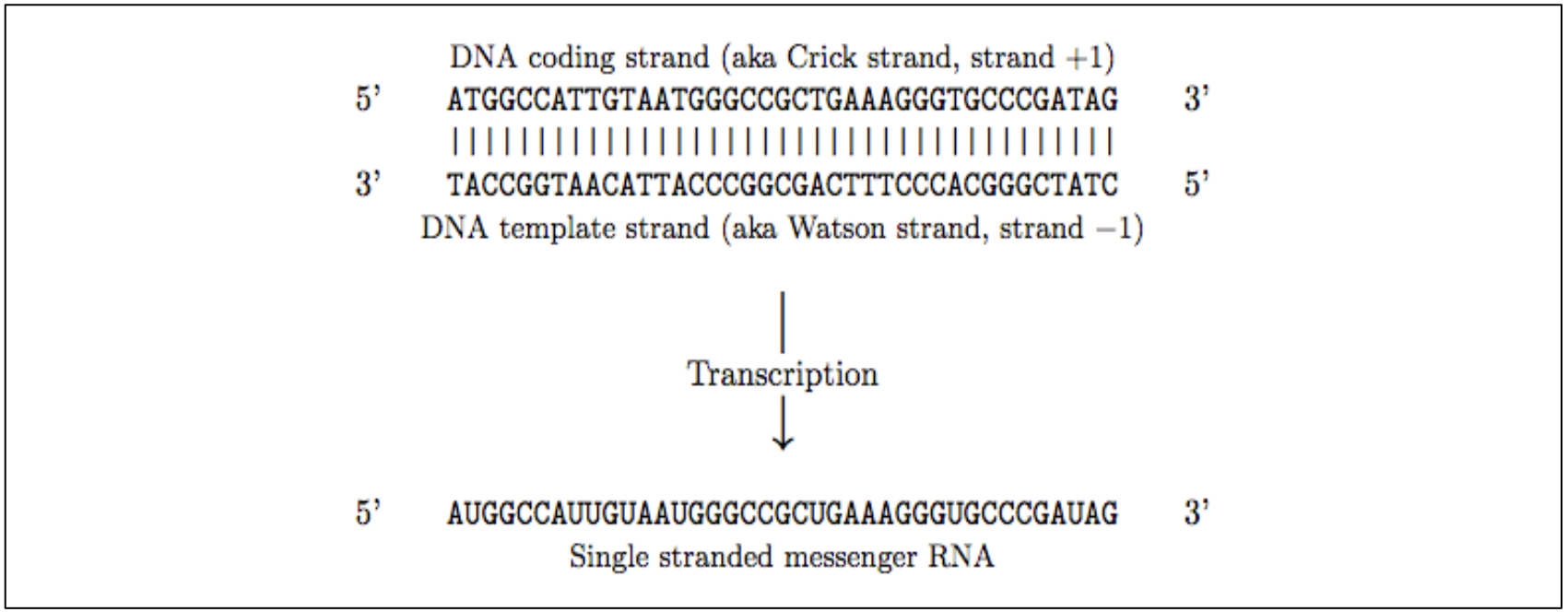
➔ For nucleotide sequences, you can easily obtain the complement or reverse complement of a Seq object using its built-in methods:

```
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPACUnambiguousDNA())
```

```
>>> my_seq.complement()
Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG', IUPACUnambiguousDNA())
>>> my_seq.reverse_complement()
Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC', IUPACUnambiguousDNA())
>>> my_seq[::-1]
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())
```

Transcription

- Consider the following:



- The actual biological transcription process works from the template strand, doing a reverse complement (TCAG -> CUGA) to give the mRNA. However, in Biopython and bioinformatics in general, we typically work directly with the coding strand because this means we can get the mRNA sequence just by switching T -> U

Transcription (2)

- Match the figure above
  - remember by convention nucleotide sequences are normally read from the 5’ to 3’ direction, while in the figure the template strand is shown reversed.

```
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> template_dna = coding_dna.reverse_complement()
>>> template_dna
Seq(‘CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT’, IUPACUnambiguousDNA())
```

- Transcribe the coding strand into corresponding mRNA, using Seq object's built in transcribe method (switch T->U and adjust the alphabet)

```
>>> coding_dna
Seq(‘ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG’, IUPACUnambiguousDNA())
>>> messenger_rna = coding_dna.transcribe()
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Transcription (3) (added in Biopython 1.49)

➔ Do a true biological transcription starting with the template strand:

```
>>> template_dna.reverse_complement().transcribe()
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousDNA())
```

➔ The Seq object also includes a back-transcription method for going from the mRNA to the coding strand of the DNA:

```
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

Translation

➔ Translate mRNA into the corresponding protein sequence

```
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

➔ You can also translate directly from the coding strand DNA sequence:

```
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

➔ Available in Biopython from the NCBI

```
>>> coding_dna.translate(table="Vertebrate Mitochondrial")
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

➔ Using the NCBI table number

```
>>> coding_dna.translate(table=2)
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

Translation (2)

- Translate nucleotides up to the first in frame stop codon, and then stop

```
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

```
>>> coding_dna.translate(to_stop=True)
Seq('MAIVMGR', IUPACProtein())
```

```
>>> coding_dna.translate(table=2)
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

```
>>> coding_dna.translate(table=2, to_stop=True) ## the stop codon itself is not translated
Seq('MAIVMGRWKGAR', IUPACProtein())
```

- complete coding sequence CDS, (e.g. mRNA { after any splicing)
- commences with a start codon, ends with a stop codon, and has no internal in-frame stop codons
- what if your sequence uses a non-standard start codon?
- This happens a lot in bacteria, for example, the gene yaaX in E. coli K12

Translation (3)

```
>>> from Bio.Alphabet import generic_dna
>>> gene = Seq("GTGAAAAAGATGCAATCTATCGTACTCGCACTTTCCCTGGTTCTGGTCGCTCCCATGGCA" + \
... "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAAATTACAGATAGGCGATCGTGAT" + \
... "AATCGTGGCTATTACTGGGATGGAGGTCACTGGCGCGACCACGGCTGGTGGAAACAACAT" + \
... "TATGAATGGCGAGGCAATCGCTGGCACCTACACGGACCGCCGCCACCGCCGCGCCACCAT" + \
... "AAGAAAGCTCCTCATGATCATCACGGCGGTCATGGTCCAGGCAAACATCACCGCTAA",
... generic_dna)
>>> gene.translate(table="Bacterial")
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HR*',
HasStopCodon(ExtendedIUPACProtein(), '*')
>>> gene.translate(table="Bacterial", to_stop=True)
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

➔ In the bacterial genetic code GTG is a valid start codon, and while it does normally encode Valine, if used as a start codon it should be translated as methionine. This happens if you tell Biopython your sequence is a complete CDS:

```
>>> gene.translate(table="Bacterial", cds=True)
Seq('MKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

Translation Tables

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_name["Standard"]
>>> mito_table = CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
```

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_id[1]
>>> mito_table = CodonTable.unambiguous_dna_by_id[2]
```

```
>>> print(standard_table)
Table 1 Standard, SGC0
```

	T	C	A	G	
T	TTT F	TCT S	TAT Y	TGT C	T
T	TTC F	TCC S	TAC Y	TGC C	C
T	TTA L	TCA S	TAA Stop	TGA Stop	A
T	TTG L(s)	TCG S	TAG Stop	TGG W	G
C	CTT L	CCT P	CAT H	CGT R	T
C	CTC L	CCC P	CAC H	CGC R	C
C	CTA L	CCA P	CAA Q	CGA R	A
C	CTG L(s)	CCG P	CAG Q	CGG R	G
A	ATT I	ACT T	AAT N	AGT S	T
A	ATC I	ACC T	AAC N	AGC S	C
A	ATA I	ACA T	AAA K	AGA R	A
A	ATG M(s)	ACG T	AAG K	AGG R	G
G	GTT V	GCT A	GAT D	GGT G	T
G	GTC V	GCC A	GAC D	GGC G	C
G	GTA V	GCA A	GAA E	GGA G	A
G	GTG V	GCG A	GAG E	GGG G	G

Translation Tables (2)

```
>>> print(mito_table)
Table 2 Vertebrate Mitochondrial, SGC1
```

	T	C	A	G	
T	TTT F	TCT S	TAT Y	TGT C	T
T	TTC F	TCC S	TAC Y	TGC C	C
T	TTA L	TCA S	TAA Stop	TGA W	A
T	TTG L	TCG S	TAG Stop	TGG W	G
C	CTT L	CCT P	CAT H	CGT R	T
C	CTC L	CCC P	CAC H	CGC R	C
C	CTA L	CCA P	CAA Q	CGA R	A
C	CTG L	CCG P	CAG Q	CGG R	G
A	ATT I(s)	ACT T	AAT N	AGT S	T
A	ATC I(s)	ACC T	AAC N	AGC S	C
A	ATA M(s)	ACA T	AAA K	AGA Stop	A
A	ATG M(s)	ACG T	AAG K	AGG Stop	G
G	GTT V	GCT A	GAT D	GGT G	T
G	GTC V	GCC A	GAC D	GGC G	C
G	GTA V	GCA A	GAA E	GGA G	A
G	GTG V(s)	GCG A	GAG E	GGG G	G

```
>>> mito_table.stop_codons
['TAA', 'TAG', 'AGA', 'AGG']
```

```
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
```

```
>>> mito_table.forward_table["ACG"]
'T'
```

Comparing Seq objects

- Meaning of the letters in a sequence are context dependent
- The letter “A" could be part of a DNA, RNA or protein sequence.
- Comparing two Seq objects could mean considering both the sequence strings and the alphabets
- Compare the sequences as string:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.ambiguous_dna)
>>> str(seq1) == str(seq2)
True

>>> str(seq1) == str(seq1)
True
```

- Sequence comparison only looks at the sequence, ignoring alphabet

```
>>> seq1 == seq2
True
>>> seq1 == "ACGT"
True
```

- Note if you compare sequences with incompatible alphabets (e.g. DNA vs RNA, or nucleotide versus protein), then you will get a warning but for the comparison itself only the string of letters in the sequence is used:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna, generic_protein
>>> dna_seq = Seq("ACGT", generic_dna)
>>> prot_seq = Seq(``ACGT", generic_protein)
>>> dna_seq == prot_seq
BiopythonWarning: Incompatible alphabets DNAAlphabet() and ProteinAlphabet()
True
```

- WARNING: Older versions of Biopython instead used to check if the Seq objects were the same object in memory.
- Important if you need to support scripts on both old and new versions of Biopython.
- Make the comparison explicit by wrapping your sequence objects with either str(...) for string based comparison or id(...) for object instance based comparison.

MutableSeq objects

- The Seq object is “read only", or in Python terminology, immutable

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
```

- Observe what happens if you try to edit the sequence:

```
>>> my_seq[5] = "G »
Traceback (most recent call last):
TypeError: 'Seq' object does not support item assignment
```

- However, you can convert it into a mutable sequence (a MutableSeq object) and do pretty much anything you want with it:

```
>>> mutable_seq = my_seq.tomutable()
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
```

- Alternatively, you can create a MutableSeq object directly from a string:

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
```

➔ Either way will give you a sequence object which can be changed:

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq[5] = "C"
>>> mutable_seq
MutableSeq('GCCATCGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.remove("T")
>>> mutable_seq
MutableSeq('GCCACGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.reverse()
>>> mutable_seq
MutableSeq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

➔ Note that unlike the Seq object, the MutableSeq object's methods like reverse_complement() and reverse() act in-situ!

```
>>> new_seq = mutable_seq.toseq()
>>> new_seq
Seq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

UnknowSeq objects

- Subclass of the basic Seq object
- Represent a sequence where we know the length, but not the actual letters making it up.

```
>>> from Bio.Seq import UnknownSeq
>>> unk = UnknownSeq(20)
>>> unk
UnknownSeq(20, alphabet = Alphabet(), character = '?')
>>> print(unk)
????????????????????
>>> len(unk)
20
```

- Specify an alphabet, meaning for nucleotide sequences the letter defaults to “N" and for proteins “X", rather than just “?"

```
>>> from Bio.Seq import UnknownSeq
>>> from Bio.Alphabet import IUPAC
>>> unk_dna = UnknownSeq(20, alphabet=IUPAC.ambiguous_dna)
>>> unk_dna
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> print(unk_dna)
NNNNNNNNNNNNNNNNNNNN
```

SeqRecord class from Bio.SeqRecord

- Allows higher level features such as identifiers and features (as SeqFeature objects) to be associated with the sequence, and is used throughout the sequence input/output interface Bio.SeqIO described later.
- Using richly annotated sequence data, say from GenBank or EMBL files,
- Cover most things to do with the SeqRecord and SeqFeature objects
- Read the SeqRecord wiki page (http://biopython.org/wiki/SeqRecord), and the built in documentation (also online SeqRecord and SeqFeature)

```
>>> from Bio.SeqRecord import SeqRecord
>>> help(SeqRecord)
...
```

The SeqRecord Object from Bio.SeqRecord module

- .seq - The sequence itself, typically a Seqobject.
- .id - The primary ID used to identify the sequence – string
  - (In most cases this is something like an accession number).
- .name – A “common" name/id for the sequence – string
  - (In some cases this will be the same as the accession number, but it could also be a clone name). analogous to the LOCUS id in a GenBank record.
- .description - A human readable description or expressive name for the sequence - string
- .letter_annotations - Holds per-letter-annotations using a (restricted) dictionary of additional information about the letters in the sequence. The keys are the name of the information, and the information is contained in the value as a Python sequence (i.e. a list, tuple or string) with the same length as the sequence itself. This is often used for quality scores (e.g. Section 20.1.6) or secondary structure information (e.g. from Stockholm/PFAM alignment les).

The SeqRecord Object (2)

- .letter_annotations - Holds per-letter-annotations using a (restricted) dictionary of additional information about the letters in the sequence. The keys are the name of the information, and the information is contained in the value as a Python sequence (i.e. a list, tuple or string) with the same length as the sequence itself. This is often used for quality scores (e.g. Section 20.1.6) or secondary structure information (e.g. from Stockholm/PFAM alignment files).
- .annotations - A dictionary of additional information about the sequence. The keys are the name of the information, and the information is contained in the value. This allows the addition of more “unstructured" information to the sequence.
- .features - A list of SeqFeature objects with more structured information about the features on a sequence (e.g. position of genes on a genome, or domains on a protein sequence). The structure of sequence features is described below in Section 4.3.
- .dbxrefs - A list of database cross-references as strings.

Creating a SeqRecord from scratch

- Usually you won't create a SeqRecord “by hand", but instead use Bio.SeqIO to read in a sequence file for you
- Create a SeqRecord

```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq)
```

- Pass the id, name and description to the initialization function or create later

```
>>> simple_seq_r.id
'<unknown id>'
>>> simple_seq_r.id = "AC12345"
>>> simple_seq_r.description = "Made up sequence I wish I could write a paper about"
>>> print(simple_seq_r.description)
Made up sequence I wish I could write a paper about
>>> simple_seq_r.seq
Seq('GATC', Alphabet())
```

Creating a SeqRecord from scratch(2)

- Identifier is very important if you want to output your SeqRecord to a file

```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq, id="AC12345")
```

- The SeqRecord has an dictionary attribute annotations

```
>>> simple_seq_r.annotations["evidence"] = "None. I just made it up."
>>> print(simple_seq_r.annotations)
{'evidence': 'None. I just made it up.'}
>>> print(simple_seq_r.annotations["evidence"])
None. I just made it up.
```

- Working with per-letter-annotations is similar, letter_annotations is a dictionary like attribute which will let you assign any Python sequence (i.e. a string, list or tuple) which has the same length as the sequence

```
>>> simple_seq_r.letter_annotations["phred_quality"] = [40, 40, 38, 30]
>>> print(simple_seq_r.letter_annotations)
{'phred_quality': [40, 40, 38, 30]}
>>> print(simple_seq_r.letter_annotations["phred_quality"])
[40, 40, 38, 30]
```

SeqRecord objects from FASTA files

- https://github.com/biopython/biopython/blob/master/Tests/GenBank/NC_005816.fna

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
SingleLetterAlphabet()), id='gi|45478711|ref|NC_005816.1|', name='gi|45478711|ref|NC_005816.1|',
description='gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus ... sequence',
dbxrefs=[])
```

```
>>> record.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG', SingleLetterAlphabet())
```

- The first word (after removing the > symbol) is used for both the id and name attributes
- The whole title line (after removing the greater than symbol) is used for the record description

```
>>> record.id
'gi|45478711|ref|NC_005816.1|'
>>> record.name
'gi|45478711|ref|NC_005816.1|'
>>> record.description
'gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus ... pPCP1, complete sequence'
```

SeqRecord objects from GenBank files

- https://github.com/biopython/biopython/blob/master/Tests/GenBank/NC_005816.gb
- Contains a single record (i.e. only one LOCUS line) and starts:

```
LOCUS NC_005816 9609 bp DNA circular BCT 21-JUL-2008
DEFINITION Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete
sequence.
ACCESSION NC_005816
VERSION NC_005816.1 GI:45478711
PROJECT GenomeProject:10638
...
```

- Contains a single record (i.e. only one LOCUS line) and starts:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=['Project:10638'])
```

- Automatically assign a more specfic alphabet

```
>>> record.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG', IUPACAmbiguousDNA())
```

SeqRecord objects from GenBank files (2)

- The name comes from the LOCUS line, while the id includes the version sufix. The description comes from the DEFINITION line:

```
>>> record.id
'NC_005816.1'
>>> record.name
'NC_005816'
>>> record.description
'Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.'
```

- GenBank files don't have any per-letter annotations:

```
>>> record.letter_annotations
{}
```

- Most of the annotations information gets recorded in the annotations dictionary, for example:

```
>>> len(record.annotations)
11
>>> record.annotations["source"]
'Yersinia pestis biovar Microtus str. 91001'
```

- The dbxrefs list gets populated from any PROJECT or DBLINK lines:

```
>>> record.dbxrefs
['Project:10638']
>>> len(record.features)
29
```

SeqFeature objects

- Attempts to encapsulate as much of the information about the sequence as possible
- based on the GenBank/EMBL feature tables
- The key idea about each SeqFeature object is to describe a region on a parent sequence, typically a SeqRecord object
- This region is described with a location object, typically a range between two positions

- .type - This is a textual description of the type of feature (for instance, this will be something like `CDS or `gene').
- .location - The location of the SeqFeature on the sequence that you are dealing with. The SeqFeature delegates much of its functionality to the location object, and includes a number of shortcut attributes for properties of the location:
  - .ref - shorthand for .location.ref - any (different) reference sequence the location is referring to (Usually just None).
  - .ref_db - shorthand for .location.ref_db - specifies the database any identifier in .ref refers to (Usually just None).
  - .strand - shorthand for .location.strand - the strand on the sequence that the feature is located on. For double stranded nucleotide sequence this may either be 1 for the top strand,1 for the bottom strand, 0 if the strand is important but is unknown, or None if it doesn't matter. This is None for proteins, or single stranded sequences.

SeqFeatures funtionalities

- .qualifiers - Python dictionary of additional information about the feature.
  - The key is some kind of terse one-word description of what the information contained in the value is about, and the value is the actual information.
  - For example, a common key for a qualifier might be \evidence" and the value might be \computational (non-experimental)." This is just a way to let the person who is looking at the feature know that it has not be experimentally (i. e. in a wet lab) con rmed. Note that other the value will be a list of strings (even when there is only one string). This is a re ection of the feature tables in GenBank/EMBL files.
- .sub_features - Represent features with complicated locations like `joins' in GenBank/EMBL files.
  - This has been deprecated with the introduction of the CompoundLocation object, and should now be ignored.

Positions and locations

- Position
  - This refers to a single position on a sequence, which may be fuzzy or not. For instance, 5, 20, <100 and >200 are all positions.
- Location
  - A location is region of sequence bounded by some positions. For instance 5..20 (i. e. 5 to 20) is a location
- FeatureLocation object
  - Need start and end coordinates and a strand
- CompoundLocation object
  - Made up of several region
- FuzzyLocation
  - Several types of fuzzy positions, so we have five classes do deal with them

Fuzzy positions

- ExactPosition
  - Represents a position which is specified as exact along the sequence. a number, and you can get the position by looking at the position attribute of the object.
- BeforePosition
  - Represents a fuzzy position that occurs prior to some specified site. In GenBank/ EMBL notation, this is represented as something like ‘<13', signifying that the real position is located somewhere less than 13. To get the specified upper boundary, look at the position attribute of the object.
- AfterPosition
  - Represents a position that occurs after some specified site. This is represented in GenBank as `>13', and like BeforePosition, you get the boundary number by looking at the position attribute of the object.

Fuzzy positions (2)

- WithinPosition
  - Models a position which occurs somewhere between two specified nucleotides. In GenBank/EMBL notation, this would be represented as '(1.5)', to represent that the position is somewhere within the range 1 to 5. To get the information in this class you have to look at two attributes. The position attribute specifies the lower boundary of the range we are looking at, so in our example case this would be one. The extension attribute specifies the range to the higher boundary, so in this case it would be 4. So object.position is the lower boundary and object.position + object.extension is the upper boundary.
- OneOfPosition
  - Occasionally used for GenBank/EMBL locations, this class deals with a position where several possible values exist, for instance you could use this if the start codon was unclear and there where two candidates for the start of the gene. Alternatively, that might be handled explicitly as two related gene features.
- UnknownPosition
  - This class deals with a position of unknown location. This is not used in Gen- Bank/EMBL, but corresponds to the '?' feature coordinate used in UniProt.

Fuzzy positions (3)

- Here's an example where we create a location with fuzzy end points:

```
>>> from Bio import SeqFeature
>>> start_pos = SeqFeature.AfterPosition(5)
>>> end_pos = SeqFeature.BetweenPosition(9, left=8, right=9)
>>> my_location = SeqFeature.FeatureLocation(start_pos, end_pos)
>>> print(my_location)
[>5:(8^9)]
```

- Note that the details of some of the fuzzy-locations changed in Biopython 1.59, in particular for BetweenPosition and WithinPosition you must now make it explicit which integer position should be used for slicing etc.
- For a start position this is generally the lower (left) value, while for an end position this would generally be the higher (right) value

```
>>> my_location.start
AfterPosition(5)
>>> print(my_location.start)
>5
>>> my_location.end
BetweenPosition(9, left=8, right=9)
>>> print(my_location.end)
(8^9)
```

Fuzzy positions (4)

- If you don't want to deal with fuzzy positions and just want numbers, they are actually subclasses of integers so should work like integers:

```
>>> int(my_location.start)
5
>>> int(my_location.end)
9
```

- For compatibility with older versions of Biopython you can ask for the nofuzzy_start and nofuzzy_end attributes of the location which are plain integers:

```
>>> my_location.nofuzzy_start
5
>>> my_location.nofuzzy_end
9
```

```
>>> exact_location = SeqFeature.FeatureLocation(5, 9)
>>> print(exact_location)
[5:9]
>>> exact_location.start
ExactPosition(5)
>>> int(exact_location.start)
5
>>> exact_location.nofuzzy_start
5
```

Keyword in

➔ See if the base/residue for a parent coordinate is within the feature/ location or not ?

```
>>> from Bio import SeqIO
>>> my_snp = 4350
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> for feature in record.features:
... if my_snp in feature:
... print("%s %s" % (feature.type, feature.qualifiers.get('db_xref')))
...
source ['taxon:229193']
gene ['GeneID:2767712']
CDS ['GI:45478716', 'GeneID:2767712']
```

➔ Note that gene and CDS features from GenBank or EMBL les defined with joins are the union of the exons - they do not cover any introns.

SeqFeature objects

Comparison

References

The format method

Parsing sequences file formats : L'objet SeqRecord

- Bioinformatics work involves dealing with the many types of file formats designed to hold biological data.

- These files are loaded with interesting biological data, and a special challenge is parsing these files into a format so that you can manipulate them with some kind of programming language.

- However, the task of parsing these files can be frustrated by the fact that the formats can change quite regularly, and that formats may contain small subtleties which can break even the most well designed parsers.
- Remember to load module SeqIO using
  - from Bio import SeqIO

Simple FASTA parsing example

➔ If you open the lady slipper orchids FASTA file ls_orchid.fasta (94 records) in your favourite text editor, you'll see that the le starts like this:

```
>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGT
GAATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGTT
GGG
```

```
>>>from Bio import SeqIO
>>>for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
        print(seq_record.id)
        print(repr(seq_record.seq))
        print(len(seq_record))
gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', SingleLetterAlphabet())
740
[...]
gi|2765564|emb|Z78439.1|PBZ78439
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', SingleLetterAlphabet())
592
```

Simple genbank parsing example

```
>>> for seq_record in SeqIO.parse("ls_orchid.gbk", genbank"):
        print(seq_record.id)
        print(repr(seq_record.seq))
        print(len(seq_record))
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
740
[...]
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
592
```

➔ Using Python iterator is within a list comprehension (or a generator expression)

```
>>> identifiers = [seq_record.id for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank")]
>>> identifiers
['Z78533.1', 'Z78532.1', 'Z78531.1', 'Z78530.1', 'Z78529.1', 'Z78527.1', ..., 'Z78439.1']
```

➔ Also “swiss” for SwissProt files or “embl” for EMBL text files

➔ See wiki page (http://biopython.org/wiki/SeqIO)

Iterating over the records in a sequence file

➔ The object returned by Bio.SeqIO is actually an iterator which returns SeqRecord objects

```
>>>record_iterator = SeqIO.parse("ls_orchid.fasta", "fasta”)
>>>first_record = next(record_iterator)
>>>print(first_record.id)
>>>print(first_record.description)
```

Z78533.1

```
>>>second_record = next(record_iterator)
>>>print(second_record.id)
>>>print(second_record.description)
```

Z78439.1

```
>>>first_record = next(SeqIO.parse("ls_orchid.gbk", "genbank"))
```

Getting a list of the records in a sequence file

- Access records in any order using Python list data type Using a list
- Much more flexible than an iterator (length of the list) but need more memory (hold all the records in memory at once).

```
>>>records = list(SeqIO.parse("ls_orchid.gbk", "genbank"))
>>>print("Found %i records" % len(records))
```

```
Found 94 records
```

```
>>>last_record = records[-1] #using Python's list tricks
>>>print(last_record.id)
>>>print(repr(last_record.seq))
>>>print(len(last_record))
```

```
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
592
```

```
>>>first_record = records[0] #remember, Python counts from zero
>>>print(first_record.id)
>>>print(repr(first_record.seq))
>>>print(len(first_record))
```

```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC',
IUPACAmbiguousDNA())
740
```

Extracting data

- As an example of how annotations are stored, we'll look at the output from parsing the first record in the GenBank file Is_orchid.gbk.
- Human readable summary of most of the annotation data for the SeqRecord

```
record_iterator = SeqIO.parse("Is_orchid.gbk", "genbank")
first_record = next(record_iterator)
print(first_record)
```

```
ID: Z78533.1
Name: Z78533
Description: C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA.
Number of features: 5
/sequence_version=1
/source=Cypripedium irapeanum
/taxonomy=['Eukaryota', 'Viridiplantae', 'Streptophyta', ..., 'Cypripedium']
/keywords=['5.8S ribosomal RNA', '5.8S rRNA gene', ..., 'ITS1', 'ITS2']
/references=[...]
/accessions=['Z78533']
/data_file_division=PLN
/date=30-NOV-2006
/organism=Cypripedium irapeanum
/gi=2765658
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
```

➔ .annotations attribute which is just a Python dictionary. Like any Python dictionary, you can easily get a list of the keys and values:

```
print(first_record.annotations)
print(first_record.annotations.keys())
print(first_record.annotations.values())
```

➔ extract a list of the species from the ls orchid.gbk GenBank le. The information we want is held in the annotations dictionary under `source' and `organism':

```
>>> print(first_record.annotations["source"])
Cypripedium irapeanum
```

```
>>> print(first_record.annotations["organism"])
Cypripedium irapeanum
```

➔ In general, `organism' is used for the scientific name (in Latin, e.g. Arabidopsis thaliana), while `source' will often be the common name (e.g. thale cress)

- In general, `organism' is used for the scientific name (in Latin, e.g. Arabidopsis thaliana), while `source' will often be the common name (e.g. thale cress)

```
from Bio import SeqIO
all_species = []
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
all_species.append(seq_record.annotations["organism"])
print(all_species)
```

```
from Bio import SeqIO
all_species = [seq_record.annotations["organism"] for seq_record in \
SeqIO.parse("ls_orchid.gbk", "genbank")]
print(all_species)
```

```
['Cypripedium irapeanum', 'Cypripedium californicum', ..., 'Paphiopedilum barbatum']
```

Parsing sequences from compressed files

Parsing sequences from the net

Parsing SwissProt sequences from the net